# Finding the Balance Between Guidance and Independence in Cybersecurity Exercises

Richard Weiss
*The Evergreen State College*

Franklyn Turbak
*Wellesley College*

Jens Mache, Erik Nilsen
*Lewis & Clark College*

Michael E. Locasto
*SRI*

## Abstract

In order to accomplish cyber security tasks, one needs to know how to analyze complex data and when and how to use a variety of tools. Many hands-on exercises for cybersecurity courses have been developed to teach these skills. There is a spectrum of ways that these exercises can be taught. On one end of the spectrum are prescriptive exercises, in which students follow step-by- step instructions to run scripted exploits, perform penetration testing, do security audits, etc. On the other end of the spectrum are open-ended exercises and capture-the- flag activities, where little guidance is given on how to proceed.

This paper reports on our experience with trying to find a balance between these extremes in the context of one of the suite of cybersecurity exercises that we have developed. The particular exercise that we present teaches students about dynamic analysis of binaries using `strace`. We have found that students are most successful in these exercises when they are given the right amount of prerequisite knowledge and guidance as well as some opportunity to find creative solutions. Our scenarios are specifically designed to develop analysis skills and the security mindset in students and to complement the theoretical aspects of the discipline and develop practical skills.

## 1 Introduction

When we choose hands-on exercises for our classes, we are faced with an apparent dilemma. Do we provide a step-by- step description with nothing left for the student to discover, or do we only describe a challenge and students must figure out what to do and how to do it? Many faculty believe that inquiry-based learning is the gold standard [1], while others are frustrated by the slow pace that this approach could entail. In addition, the structure of many classes does not allow students to explore the material at their own rate. Moreover, some researchers, including the keynote speaker at the 2016 Computer Science Education conference SIGCSE [17], stress the importance of explicit instruction. We have attempted to find a balance between these two extremes.

Hands-on exercises provide an opportunity for differential instruction. To the extent that instructors have multiple exercises to choose from and exercises that are flexible, they can decide how much guidance to give students. Note that difficulty and degree of guidance are related but not identical parameters for instruction. Difficulty has to do with the level of abstraction and the complexity of a system that is being described. Guidance has to do with the size of the steps from the users current level of knowledge or understanding to that which is required to complete the exercise. Often, learning a concept or knowledge area is broken down into multiple levels of difficulty. Students can progress from one level to the next once they have mastered the skills and concepts of the former. The degree of guidance has to do with the hints, clues, examples, and explanations that are given to the student, as well as the difference in difficulty of successive levels. In their Usenix presentation, Chung and Cohen [5] discuss the escalation in difficulty of challenges in the CSAW CTF and the fact that students can become discouraged when faced with difficult challenges in a competition[1]. Feng also claims that CTF competitions that are unguided are often discouraging and frustrating for beginners and do not promote steady learning [11].

One of our primary educational goals in creating exercises is to nurture analysis skills. When speaking of analysis skills, we mean the ability to reason about large, complex, and opaque data and systems. Strong analytical skills enable people to impose structure and meaning on such artifacts, reason about these relationships, and draw meaningful conclusions or inferences. These

[1] https://www.youtube.com/watch?v=57oyVmMYhwI

are precisely the kinds of skills that we believe are useful in many cybersecurity scenarios, from security policy design to reverse engineering to vulnerability analysis. Analysis skills complement the security mindset, which is the ability to think about how systems can fail, and be made to fail in different ways, even as one is designing a system. Questioning assumptions plays an important role in both defense and attack. In designing our exercises, we focus on the following analysis skills:

1. Verifying assumptions by checking network messages, protocols, file formats and other input data constraints to see if layers of abstraction are coherent and correct; enumerating and checking if failure modes, exceptions, and errors are controlled, caught or anticipated.

2. Gaining understanding of program, network, or system behavior and semantics, network topology or organization, or a defense posture; observing and enumerating how software components or network elements are actually composed.

3. Extracting Information from opaque artifacts. For example, analyzing a raw dump of network traffic, intrusion alerts, firewall logs, system call traces, or executable files and recognizing anomalies in a mass of otherwise normal data.

4. Creating Emergent Resilience by understanding a system well enough to design and propose enhancements to reliability, fault tolerance, or availability.

Most students, and even many computer science faculty, do not have these skills or the prerequisite knowledge to distinguish normal behavior from abnormal. Our goal has been to develop exercises that will cultivate this ability, whether it is applied to reverse engineering or penetration testing or network intrusion detection. We have developed a number of hands-on cybersecurity scenarios that have been tested in the classroom. Some results for one of these exercises that addresses network reconnaissance has been presented elsewhere [20]. In this paper, we report on a reverse engineering scenario which has raised issues of guidance for us as faculty. The scenario has been modified several times, and in our most recent version, we have surveyed our students about the perceived level of guidance.

## 2 Related Work

Our philosophy on information security education stems from our understanding and teaching of the hacker curriculum as described by Bratus [2]. This approach is predicated on the utility of understanding failure modes. Rather than teaching students the success cases, we attempt to deliver a culture shock that makes them disrespect API boundaries and adopt a cross-layer view of the CS discipline as described by Bratus et al. [4]. We also routinely encourage our students to adopt a dual frame of mind (attacker and defender) when solving problems to prevent artificial abstraction layers from becoming boundaries of competence [21]. The importance of analysis skills as explained by Bratus et al. [3] is based on linking expected behavior to actual behavior as seen in network traces, log files, program binaries, rules/policies, system call traces, network topologies, network interactions, unknown protocols, injected backdoor code, etc. All of our exercises are based on skills like these. A tool that actually supports this type of analysis is NetCheck [23], which is used to debug network applications. Using a simplified model of normal network behavior, NetCheck collects information about network applications using `strace` and compares it to the model.

There are a number of well-known lab exercises that are very prescriptive. They give explicit instructions to the student on what to do and how to do it and they don't require the students to do a deep analysis in order to answer the questions. For example, Towsons Security Injections [18] mainly focus on several important secure programming patterns, but do not emphasize analysis. The SEED [9] project presents a mature, well-documented set of exercises, which are not typically interactive or dynamic and require significant work to set up and run. They are very prescriptive in terms of their description of what the student does and they require extensive knowledge. Some of them were designed for the graduate level. However, there is nothing in the lab that requires students to analyze what the software system is doing. The Principles of Computer Security lab manual written by Nestler et al [16] provides a broad overview of cyber security, yet is also prescriptive. The exercises described by Yuan et al [23] seem to emphasize tools for auditing software, but not the analysis skills.

In contrast, open-ended cybersecurity games and capture-the- flag competitions are known to engage students [19, 12]. This includes competitions such as CCDC[2], Plaid[3], notsosecure[4], iCTF[5] [8], CSAW[6] [12], TRACER FIRE[7], Packetwars[8], and many others. These activities often provide little guidance. Because they require a significant amount of infrastructure and preparation by the organizers, they only reach a small number of

[2] `http://nationalccdc.org`
[3] `http://www.pwning.net`
[4] `http://ctf.notsosecure.com`
[5] `http://ictf.cs.ucsb.edu`
[6] `https://csaw.isis.poly.edu`
[7] `http://csr.lanl.gov/tf`
[8] `http://packetwars.com`

students. Some competitions such as CCDC and Packetwars require the installation of physical hardware, and they often require that students and their faculty travel to participate. However, they are moving to virtual environments and qualifying rounds for CCDC are run remotely. There are also a number of non-technical games with the goal of interesting students with no technical background in cybersecurity. These include Control-Alt- Hack [7], d0x3d! [13], Security Cards[9], CyberCIEGE[10] [6] and Werewolves [10]. The last of these introduces players to the concept of covert channels in a non-technical context. Online exercises that are not necessarily competitions but provide challenging exercises would include Google Gruyere[11] and overthewire[12]. Our exercises are intended to create scenarios that are closer technically to real-world situations that a security professional would face. We want exercises that our students can use in the classroom and even as training for some of these competitions if they are really attracted to them. We view the exercises described in the next section as a middle ground between the two ends of the spectrum.

There has been some research to try to determine whether guided or unguided instruction is better. Kirschner, Sweller and Clark [14] have argued that minimally guided instruction is not effective. One thing to keep in mind is that they are focusing on content knowledge alone and not skills or abilities. On the other side, Kussmal et al [15] have shown positive results with inquiry-based learning in computer science. Wolfman and Bates [22] have written about the advantages of kinesthetic learning in the classroom. These viewpoints are consistent with our observations, which are that something between the two extremes works well. We assume that in cyber security, the student needs to acquire content knowledge, skills using tools and analysis skills (abilities).

## 3 Description of the `strace` Scenario

For this paper, we focus on degree of guidance that was needed by the students for a scenario that consists of multiple exercises using the `strace` tool to examine system calls, what they learned in this scenario, and how we modified the scenario in response to feedback from the students. The `strace` scenario has been used by several faculty at different schools and has undergone multiple revisions over the last two years. `strace` is a Linux tool that generates a trace of all the system calls made during the execution of a program. These include memory, file, process, and networking operations. It is used to analyze the runtime behavior of programs, especially executables for which there is no source code. In the context of cybersecurity, `strace` is useful for detecting if a program is doing something unusual and potentially malicious, such as reading information from unexpected sources, writing information to an obscure file, surprisingly forking a child process, or performing suspicious network communication.

There are multiple goals for this exercise. One of them is for students to learn how to use `strace` to analyze what a program is doing. Another is to understand the operation of normal programs with respect to system calls and detect abnormal behavior in malicious programs. As with many tools for security analysis, effectively using `strace` to detect abnormal behavior requires (1) sifting through a large amount of data and (2) being able to distinguish abnormal behavior from normal behavior. For example, the output of `strace` when running the empty C program compiled from

```
int main () {}
```

has 23 lines involving 11 different system calls, many of which (e.g., `brk`, `ftstat64`, `mmap2`, `munmap`, `mprotect`) only make sense to those with some background in the Linux operating system. This output summarizes system calls made when running any program. For malware analysis, students must learn what to ignore in order to focus on calls made by the program itself. So the `strace` scenario begins with an exercise on studying the output of `strace` running on the empty program. However, it is important to let students know not to focus on understanding all of the system calls. We found that without that guidance, students readily explored paths that did not help them solve the challenges of the scenario or meet the learning goals.

In the next part of the scenario, students are asked to explain the output of `strace` for a C program that performs a character-by-character copy of an input file to an output file. This introduces them to system calls for opening and closing files and reading and writing file information, and they can correlate aspects of the dynamic execution reported by `strace` with static features of the program. It also exposes them to input/output buffering performed by the system; although the source code copies the file one character at a time, the `read` and `write` calls reported by strace manipulated bigger strings.

Running `strace` on even simple programs can easily generate hundreds or thousands of lines of output, so it is important for students to learn ways to filter and summarize this information. There are options for counting the number of calls made to different system routines and summarizing the time spent in them as well as options for showing only certain calls or categories of calls (e.g., file

[9] `http://securitycards.cs.washington.edu`
[10] `http://www.cisr.us/cyberciege`
[11] `https://google-gruyere.appspot.com`
[12] `http://overthewire.com`

operations, process management operations, networking operations). By default, `strace` does not trace calls of child processes forked from the main process, but there is a `-f` option for doing this. This is important to know, because malicious code often create new child processes.

The exercises introduce some of these options. Once they have learned the basics of `strace`, students are asked to use it to analyze some executables for which they have no source code. These exercises culminate in the trojaned `cat` exercise, where the usual Linux `cat` command for displaying file contents has been replaced by a trojaned version that additionally writes the contents of every displayed file to a special directory. `strace` exposes the operations that open, write to, and close the file in the special directory.[13]

The `strace` scenario has been tested several times starting in Fall 2014 in a computer security course at a liberal arts college. There were 29 students in this class who worked on the exercises in groups of two or three over two 70-minute class sessions. Initially, the instructor circulated around the room while students worked on the `strace` exercises from a handout, observing what they were doing and answering questions. It soon became apparent that most students did not understand the purpose of examining the output of `strace` on the empty program. None of the students had operating systems experience, and many were trying to understand the details of system calls like `brk` and `mmap2`. The instructor stepped in to explain that the point of this exercise was to show that many system calls are made to run any program, and that these form a boilerplate that can be ignored in subsequent operations. Similarly, for the copy exercise, the instructor needed to emphasize that the point of the exercise was to relate file operations from the dynamic trace to lines of the program. By the end of the second session, most groups had spent significant time on the trojaned `cat` problem, and most had discovered that the program was surreptitiously squirreling away copies of the displayed files. This test of the `strace` scenario highlighted issues involving the level of guidance and independence in hands-on security exercises. We have included the questions and some sample student answers in the Appendix.

The scenario was repeated at the same college in Spring 2016 with 23 students. This time the instructor went over the first two questions as a demo to emphasize that the point of these questions was what to *ignore* in the output of `strace`. Students worked in groups on the other problems except for the trojaned `cat` question, which was assigned as an individual homework problem. Even though the in-class exercises had emphasized that boilerplate system calls should be ignored and had exposed students to `strace` options for viewing forked processes and filtering the output for file operations, many students still got sidetracked with the complexities of unfiltered `strace` output, or did not generate output for forked processes (which was essential to solving the problem).

The scenaerio was also run a another liberal arts college in Spring 2016, where the instructor gave more guidance. The first five parts of the scenario were done by the instructor as a demo, and the remaining parts were done by the students. The students were surveyed on their perceived level of guidance. Using a 5-point Likert scale, they indicated whether they wanted more guidance or more independence. Out of 10 students who responded, 7 wanted more guidance, but the responses included the full range, indicating that finding a balance of guidance and independence that works for all students seems unlikely.

## 4 Discussion and Lessons Learned

We have considered and tried a number of strategies in the context of the `strace` scenario, and here are some of them:

- Does the instructor tell the students how the tool works, what it can do, what to pay attention to and what to ignore? Or are these left for the student to discover by running simple examples? We tried both, and students seem to work more efficiently when told what to pay attention to.

- Does the instructor present students with a general problem first, or show them an example? The instructor can choose the level of guidance. Our early experience was that students could be distracted by investigating what the empty program actually does vs. moving on to examine programs that do more interesting things. Similarly, students could get caught up reading about all of the different system calls and lose sight of the goal of finding anomalous behavior. In the most recent trial, the instructor gave a demo of the first steps in the scenario, while the students followed along. This saved the students some time.

- When it comes to understanding options for a tool like `strace`, there is a tradeoff between having students explore options based on documentation (e.g., `man` pages) and having an instructor show particularly useful options. The `man` page approach fosters

[13] In the initial version of the scenario, the file contents displayed by `cat` were appended (along with the file name) to a single file named `/tmp/carnivore`. However, this led to undesirable behavior when a student used the trojaned `cat` to display the contents of `/tmp/carnivore` — the file would double in size! So in subsequent versions of the scenario, the file contents displayed by each call to `cat` were written to a separate file file in the directory `/tmp/data`.

independence in the teach-a-person-to-fish kind of way; but it can take significant time, and some students may not find options they can use for effective analysis. A middle ground is to have the students interactively experiment with options during a relatively short portion of the class and then have a discussion about which options are particularly useful and why.

- Tools like `strace` involve several levels of knowledge. In order to be able to make sense of simple traces, students need to know something about system calls and their relationship to programs. They must also understand tool options for controlling the amount of information and summarizing it. These basic kinds of knowledge are a prerequisite to performing higher-level analysis, such as distinguishing normal from abnormal behavior, and associating abnormal behavior with particular kinds of malware.

- Exercises need to be carefully written to explain the purpose of the exercise and what students should and should not focus on, and to constrain the exploration of the students. Otherwise, students can spend much of their time possibly unproductively exploring blind alleys. The exercises have questions, as shown in the Appendix, to guide the students and assess whether they have understood the high-level purpose of the exercise, plus details relevant to subsequent exercises. However, we discovered that in subtle ways, our exercises were not always aligned with our objectives.

- When students have limited background knowledge (of operating systems, in this particular case), it can be helpful for an instructor to give a high-level overview of the area and explain what is and is not important about the details. Otherwise, students can spend significant time trying to understand details that are not important.

- In the context of undergraduate colleges and community colleges, it is not uncommon for security courses to be taught by instructors with limited background in the area. So instructors will be learning how to use tools one step ahead of, or alongside, the students. Exercises need to be designed so that instructors are encouraged to modify them. By thinking about how to modify the exercises, the instructors gain more confidence and a deeper understanding of the content. We found this to be an advantage in `strace`. However, the metacognitive level of the instructors was higher than that of the students, which made it more difficult for them to anticipate the students' questions and confusions.

Different students need different amounts of guidance, and different instructors will make different decisions about the level of scaffolding and guidance that they want to provide. We are not claiming that there is only one way to teach analysis skills and the security mindset. What is important from the perspective of designing hands-on exercises is providing flexibility to the instructor. There is a broad spectrum of hands-on exercises that are available. Some are very prescriptive and limited in scope while some are so open-ended that the student is not even told what to look for. We have found with our students that a middle ground seems to work well. We avoid being too prescriptive in terms of giving a recipe for what to do, but we give significant guidance through the questions that are included with each exercise. One of the strategies we tried was presenting some of the exercises as a demo with students following. Another approach that we plan to use is letting students work on one of the exercises for a short period of time and then have the instructor lead a discussion.

Some of the questions that we used to frame the scenario are shown in the Appendix. For example, Q1 describes exactly what students should do, which is very prescriptive, but the question is very open-ended. In this case, by talking with the students during the exercise, the instructors found that the question was not direct enough, and they were able to provide more guidance in response.

Q6 (the trojaned `cat` question) seemed to work very well, judging by the student answers. By the time they answered Q6, the students had had more experience with `strace` and were looking for anomalous behavior. In contrast to Q1, the instructions are less prescriptive, but the questions are more precise. Most of the students gave answers to this question that showed that they understood how to use the tool and how to find anomalous behavior. The two students answers shown in the Appendix are representative and indicate that the students were not just answering the focused questions, but were implicitly posing their own.

In contrast, we saw problems when we used other hands-on exercises, e.g. a man-in-the-middle attack, that were very prescriptive in terms of procedure and the questions were precise but did not address the learning objectives, e.g. listing IP addresses and ports. In those cases, when we talked with students as they were doing the labs, they generally did not understand the principles behind what they were doing.

We have also found that having a precise record of what students did during the exercises can be very useful for giving them feedback. In the Spring 2016 iteration of the `strace` scenario, we captured `bash` histories from the student accounts to get a detailed picture of what commands they executed. This helps the instructor understand problems encountered by students that may

not be apparent from their writeups for the questions. In one of the classes, summaries of the `bash` histories were even used as a basis for discussion in class after the scenario was completed. We are experimenting with visualizations of the `bash` histories, and how to incorporate discussions of such visualizations into the learning process of our security scenarios.

## 5 Conclusions

When designing hands-on exercises, there are trade-offs that need to be taken into account. In theory, independent discovery is great. It has the power to engage the creative energy of the student, and can lead to deeper understanding, but that comes at a price. Independent study can require a significant amount of time from the students and the instructors. Students may miss what the goals of the exercise are and may be frustrated or lost. This tension could put demands on the instructor given that different students are going in different directions. We have found that what seems to work well in our example is coupling scenarios that are less prescriptive in terms of telling students what to do, with providing guidance in terms of the questions that they ask. We have found that the `strace` exercise and others that we created have significant potential for students to discover their own solutions and develop analysis skills when provided a reasonable amount of guidance.

Cyber security is a field that connects with almost all other topics in Computer Science, yet we ought not teach it only when students have all of the prerequisite knowledge. We have used our exercises in several classes at 4-year colleges that serve a diverse student population in terms of prerequisite knowledge and learning rates. As a consequence, we have learned to fill in the gaps in students preparation with lectures, demos, discussions, and guidance during the exercises. Our experiences and the experiences of our students have been very positive. In addition, the feedback we have received from other faculty who have tried our exercises, is that their interest in cyber security has increased.

## 6 Acknowledgments

This work was supported in part by the National. Science Foundation under grants 1516100, 1516730, 1141314, and 1141341.

`http://edurange.org`

## References

[1] BELL, T., URHAHNE, D., SCHANZE, S., AND PLOETZNER, R. Collaborative inquiry learning: Models, tools, and challenges. *International Journal of Science Education 32*, 3 (2010), 349–377.

[2] BRATUS, S. What hackers learn that the rest of us don't: Notes on hacker curriculum. *IEEE Security and Privacy 5* (2007), 72–75.

[3] BRATUS, S., D'CUNHA, N., SPARKS, E., AND SMITH, S. W. Toctou, traps, and trusted computing. In *Trusted Computing-Challenges and Applications*. Springer, 2008, pp. 14–32.

[4] BRATUS, S., SHUBINA, A., AND LOCASTO, M. E. Teaching the principles of the hacker curriculum to undergraduates. In *Proceedings of the 41st ACM technical symposium on Computer science education* (New York, NY, USA, 2010), SIGCSE '10, ACM, pp. 122–126.

[5] CHUNG, K., AND COHEN, J. Learning obstacles in the capture the flag model. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)* (2014).

[6] CONE, B. D., IRVINE, C. E., THOMPSON, M. F., AND NGUYEN, T. D. A video game for cyber security training and awareness. *Computers & Security 26*, 1 (2007), 63 – 72.

[7] DENNING, T., LERNER, A., SHOSTACK, A., AND KOHNO, T. Control-alt-hack: the design and evaluation of a card game for computer security awareness and education. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 915–928.

[8] DOUPÉ, A., EGELE, M., CAILLAT, B., STRINGHINI, G., YAKIN, G., ZAND, A., CAVEDON, L., AND VIGNA, G. Hit'em where it hurts: a live security exercise on cyber situational awareness. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 51–61.

[9] DU, W., AND WANG, R. Seed: A suite of instructional laboratories for computer security education. *Journal on Educational Resources in Computing (JERIC) 8*, 1 (2008), 3.

[10] ENSAFI, R., JACOBI, M., AND CRANDALL, J. R. Students Who Don't Understand Information Flow Should Be Eaten: An Experience Paper. In *Proceedings of the 5th USENIX conference on Cyber Security Experimentation and Test (CSET'12)* (2012), pp. 10–10.

[11] FENG, W.-C. A scaffolded, metamorphic ctf for reverse engineering. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)* (2015).

[12] GAVAS, E., MEMON, N., AND BRITTON, D. Winning cybersecurity one challenge at a time. *Security & Privacy, IEEE 10*, 4 (2012), 75–79.

[13] GONDREE, M., AND PETERSON, Z. N. Valuing security by getting [d0x3d!]: Experiences with a network security board game. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test* (Berkeley, CA, 2013), USENIX.

[14] KIRSCHNER, P. A., SWELLER, J., AND CLARK, R. E. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist 41*, 2 (2006), 75–86.

[15] KUSSMAUL, C. Process oriented guided inquiry learning (pogil) for computer science. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2012), SIGCSE '12, ACM, pp. 373–378.

[16] NESTLER, V., HARRISON, K., HIRSCH, M., AND CONKLIN, W. A. Principles of computer security lab manual.

[17] SWELLER, J. Cognitive load theory and computer science education. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (New York, NY, USA, 2016), SIGCSE '16, ACM, pp. 1–1.

[18] TURNER, C. F., TAYLOR, B., AND KAZA, S. Security in computer literacy: A model for design, dissemination, and assessment. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2011), SIGCSE '11, ACM, pp. 15–20.

[19] VIGNA, G. Teaching Network Security through Live Exercises. In *Proc. 3rd Ann. World Conf. Information Security Education (WISE 03)* (2003), Kluwer Academic, pp. 3–18.

[20] WEISS, R., LOCASTO, M. E., AND MACHE, J. A reflective approach to assessing student performance in cybersecurity exercises. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (2016), ACM, pp. 597–602.

[21] WHITE, G., AND NORDSTROM, G. Security across the Curriculum: Using Computer Security to Teach Computer Science Principles. In *Proceedings of the* $19^{th}$ *National Information Systems Security Conference* (1996), NIST, pp. 483–488.

[22] WOLFMAN, S. A., AND BATES, R. A. Kinesthetic learning in the classroom. *J. Comput. Sci. Coll. 21*, 1 (Oct. 2005), 203–206.

[23] ZHUANG, Y., GESSIOU, E., PORTZER, S., FUND, F., MUHAMMAD, M., BESCHASTNIKH, I., AND CAPPOS, J. Netcheck: Network diagnoses from blackbox traces. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), USENIX*.

[23] Yuan, X., Hernandez, J., Waddell, I, Chu, B, and Yu, H., Hands-on Laboratory Exercises for Teaching Software Security, Proceedings of the 16th Colloquium for Information Systems Security Education, Lake Buena Vista, Florida, June 11 - 13, 2012

## Appendix: `strace` Exercises

*Q1:* Your home directory in the account on the EDURange `strace` scenario NAT instance contains various files that will be used in this scenario. One is the file `empty.c`, whose contents is:

```
int main () {}
```

Compile this and run `strace`. What do you think the output of `strace` indicates in this case? How many different system call functions do you see?

*Q2:* The `-o` option of `strace` writes its output to a file. Do the following:

```
strace -o empty1 ./empty
strace -o empty2 ./empty
diff empty1 empty2
```

Explain the differences reported between traces `empty1` and `empty2`.

*Q3:* Study the following program `copy.c`.

```
# include <stdio.h>
# include <stdlib.h>
int main (int argc, char** argv) {
  char c;
  FILE* inFile;
  FILE* outFile;
  char outFileName[256];
  if (argc != 3) {
    printf("program usage: ./copy <infile> "
    printf("<outfile>\n");
    exit(1);
  }
  snprintf(outFileName, sizeof(outFileName),
   "%s/%s", getenv("HOME"), argv[2]);
  inFile = fopen(argv[1], "r");
  outFile = fopen(outFileName, "w");
  printf("Copying %s to %s\n", argv[1],
   outFileName);
  while ((c = fgetc(inFile)) != EOF) {
    fprintf(outFile, "%c", c);
  }
  fclose(inFile);
  fclose(outFile);
}
```

Compile it to an executable named `copy`, and use `strace` to execute it. Explain the non-boilerplate parts of the trace by associating them with specific lines in `copy.c`.

*Q4:* The file `strace-identify` was created by calling `strace` on a command. The first line of the trace has been deleted to make it harder to identify. Determine the command on which `strace` was called to produce this trace.

*Q5:* The file `mystery` is an executable whose source code is not available. Use `strace` to explain what the program does in the context of the following examples:

```
mystery foo abc
mystery foo def
mystery baz ghi
```

*Q6: Note: please do Q7 and Q8 before this problem; it turns out theyre helpful for solving this problem.*[14] Create a one-line secret file. Heres an example, though of course you choose something different as your secret:

```
echo "My phone number is 123-456-7890" > secret
```

Now display the secret to yourself using cat:

```
cat secret
```

Is your file really secret? How much do you trust the `cat` program? Run `strace` on `cat secret` to determine what its actually doing. Based on this and subsequent experiments, determine answers to the following questions:

[14] In the Fall 2014 version of the `strace` scenerio, questions Q7 and Q8 were added after an initial version of the exercises had been published, so they were put at the end. In some subsequent versions, the questions were reordered so that the trojaned `cat` question became the last one.

1. Does the `cat` program in the `strace` scenario do more than display the contents of a file? Exactly what else does it do?

2. How can you display the contents of a file without the extra actions reported above?

3. Can anyone else read your secret?

4. Can you read the secrets of anyone else?

5. How do you think the Trojaned `cat` program was implemented? How do you think it was installed? Justify your explanations.

Answering these questions will require some careful forensics work on your part. Write up all answers in your solution doc. Explain all experiments you perform, the key results of those experiments, what you learned from each experiment, and hypotheses that you developed along the way. Notes:

- Your writeup should clearly answer all the asked questions and include sufficiently detailed explanations, evidence, etc.

- Include particular lines from the output of `strace` in your explanations where they are relevant.

- Include transcripts with the output of Linux commands in your explanations when they are relevant.

- Note that the `strace` scenario does not include the `emacs` editor. If you want to use an editor to read or modify a file, you can use the `nano` or `vim` editors.

- Don't incorrectly assume that just because you can't list the files in a directory that you can't read the contents of particular files in that directory.

*Q7:* Here is a simple shell script in `script.sh`:

```
#!/bin/bash
echo "a" > foo.txt
echo "bc" >> foo.txt
echo `id -urn` >> foo.txt
chmod 750 foo.txt
/bin/cat foo.txt | wc
```

Compare the outputs of the following calls to `strace` involving this script. Explain what you see in the traces in terms of the commands in the script.

```
./script.sh
strace ./script.sh
strace -f ./script.sh
```

*Q8:* Sometimes `strace` prints out an overwhelming amount of output. One way to filter through the output is to save the trace to a file and search through the file with grep. But `strace` is equipped with some options that can do some summarization and filtering. To see some of these, try the following, and explain the results:

```
find /etc/pki
strace find /etc/pki
strace -c find /etc/pki
strace -e trace=file find /etc/pki
strace -e trace=open,close,read,write find /etc/pki
```

**Sample Student Answers to Q6:**

*Student1:* Our file isn't very secret; don't trust `cat`. The `cat` we are using isn't `/bin/cat`, so it is acting differently. Everything goes into the `/tmp/carnivore` file. To search large carnivore, you can `grep` it for student and print both the student line and the line following it (the password), which is `grep -A 1 student carnivore`. Using `which cat`, we can see that the cat we are using is the result of a path attack. To get rid of the trojan `cat`:

```
export PATH=/bin:/usr/bin:/usr/local/sbin:
/usr/sbin:/sbin:/opt/aws/bin:/home/student_xx/bin
```

This resets your `PATH` and gets rid of `/usr/local/bin` as a possible path.

*Student2:* ... It then resumes a normal `cat` operation. In `/tmp/carnivore` though, other students can read the secret AND you can read other students secrets since they are being stored in a directory called `tmp` that has `drwxrwxrwt` permissions. Found this article to refresh ourselves on what the sticky bit `t` meant: ...